DOCUMENT RESUME

ED 258 805 SE 045 801

AUTHOR Bonar, Jeffrey; Soloway, Elliot
TITLE Pre-Programming Knowledge: A Major Source of Misconceptions in Novice Programmers.
INSTITUTION Pittsburgh Univ., Pa. Learning Research and Development Center.
SPONS AGENCY National Science Foundation, Washington, D.C.; Office of Naval Research, Washington, D.C.
PUB DATE May 85
CONTRACT N00014-83-6-0148; N00014-83-K-0655
GRANT NSF-G-MCS-8302382
NOTE 48p.
PUB TYPE Reports - Research/Technical (143)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Computer Science Education; Computer Software; Educational Research; Higher Education; *Knowledge Level; *Models; *Programing
IDENTIFIERS *Misconceptions; National Science Foundation

ABSTRACT

A process model to explain bugs produced by novices early in a programming course are presented. (The term "bug" refers to an error in a person's behavior, particularly an error in a computer program they have written.) The model is motivated by interviews of novice programmers solving simple programming problems. The key idea is that many programming bugs can can be explained as novices inappropriately using their knowledge of step-by-step procedural specifications in natural language. (The term "natural language" refers to the language in which step-by-step procedures are written.) Programming bugs are viewed as patches generated in response to an impasse reached by the novice while developing a program; such patching strategies are called bug generators. Several bug generators are used to describe how natural language pre-programming knowledge is used by novices to create patches. Other kinds of bug generators are also discussed. A representation both for novice natural language pre-programming knowledge and novice fragmentary programming knowledge is described. Using these representations and the bug generators, the model is evaluated by analyzing four interviews with novice programmers. (JN)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED258805

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
☑ This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

Pre-Programming Knowledge: A Major Source of Misconceptions in Novice Programmers

Jeffrey Bonar and Elliot Soloway

May 1985

Jeffrey Bonar
Intelligent Tutoring Systems Group
Learning Research and Development Center
University of Pittsburgh
Pittsburgh, PA 15260

Elliot Soloway
Cognition and Programming Project
Computer Science Department
Yale University
New Haven, CT 06520

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
J. Aug
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Copyright © 1985 Jeffrey Bonar and Elliot Soloway

To Appear in *Human-Computer Interaction*, Fall 1985.

The paper is based on the PhD thesis of the first author.

This work was supported by the National Science Foundation under NSF Grant MCS-8302382. Currently, the first author is supported by the Office of Naval Research under Contract Numbers N00014-83-6-0148 and N00014-83-K-0655. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the authors, and do not necessarily reflect the views of the U.S. Government.

SE 045 801

Table of Contents

1. Introduction 1

2. Representing Novice Programming Knowledge 5
- 2.1. SSK: Pre-Programming Knowledge 5
- 2.2. PK: Fragments of Programming Knowledge 6
- 2.3. Functional and Surface Links Between SSK and PK 10

3. The Process of Generating a Bug 11

4. Evaluating the Model 19
- 4.1. Methodology 20
- 4.2. Overview of Protocols Analyzed 22

5. Results of the Analysis 28
- 5.1. Expectation 1: Regular and Extensive Use of Plans 28
- 5.2. Expectation 2: Bug Generators Plausibly Explain the Bugs 31
- 5.3. Expectation 3: SSK/PK Bug Generators Are Critical 34

6. Concluding Remarks 34

Acknowledgments 39

Bibliography 39

I. A Sample of an Analyzed Protocol 41

Abstract

We present a process model to explain bugs produced by novices early in a programming course. The model is motivated by interviews of novice programmers solving simple programming problems. Our key idea is that many programming bugs can be explained as novices inappropriately using their knowledge of step-by-step procedural specifications in Natural Language. We view programming bugs as patches generated in response to an impasse reached by the novice while developing a program. We call such patching strategies bug generators. Several of our bug generators describe how Natural Language pre-programming knowledge is used by novices to create patches. Other kinds of bug generators are are also discussed. We describe a representation both for novice Natural Language pre-programming knowledge and novice fragmentary programming knowledge. Using these representations and the bug generators, we evaluate the model by analyzing four interviews with novice programmers.

1. Introduction

There is a growing literature about bugs[1] in novice programming (Shneiderman, 1976; Johnson et al., 1982; Kahney & Eisenstadt, 1982; Anderson et al., 1984; Soloway & Ehrlich, 1984) . In this article we propose a process model to explain bugs produced in early phases of an introductory course. We are concerned with the difficulties a novice has in using basic programming constructs in short programs. In particular, we focus on the knowledge that novice programmers bring to these early programming problems. Our key idea is:

> ***Many novice programming bugs can be explained as inappropriate use of the knowledge used in writing step-by-step procedural specifications in Natural Language.***[2]

Our model is motivated by patterns of behavior we repeatly observed in video-taped interviews of novice programmers solving programming problems (see Bonar (1985) for a complete discussion). In particular, we characterize what happens when a novice produces a bug:

---

[1] There are two usages for the term "bug". We are using it to refer to an error in a person's behavior, particular an error in a computer program they have written. Often though, "bug" is used to refer to perturbations in a person's mental procedure for some task. In this usage, systematic errors in behavior are explained by the bugs in mental procedures. This second meaning is used by Brown and VanLehn (1980) and Resnick (1982) when discussing children's problems with multi-column subtraction.

[2] Throughout, the term *Natural Language* will be used to refer to the language in which step-by-step procedures are written. "English", the other obvious choice, was not used because it unnecessarily implied that the novice programming phenomena discussed here were limited to English.

Note that these procedures are not strictly step-by-step, though they often take the form of a numbered list of statements. In actual usage, a number might label one step, several steps, or even a "step" that actually modifies one or more previous numbered statements or establish a global condition (demon).

- While solving a programming problem (writing a program), novices will encounter some aspect of the problem they don't understand - an impasse.
- In order to move beyond the impasse, novices cast about for a way to resolve the aspect of the problem they don't understand - a patch. Frequently that resolution involves an appeal to their knowledge of of Natural Language step-by-step procedures that would be applicable in a similar situation.
- In implementing the patch, a bug is introduced.

Consider an example from one of our interviews. Subject 13 is working on the Averaging Problem (Figure 1). This problem requires the student to read in a series of numbers, watching for an ending value of 99999, and producing the average of the numbers read in before the 99999. Producing the average requires the student to accumulate both a sum and count of the numbers read. In the excerpt of Figure 2 he has just specified the test for a `repeat until` loop and is considering how to implement the step after the loop. (Note that the most straightforward solution to this problem would use a `while` loop rather than the `repeat until` loop. We will not focus on that issue here, but see Soloway et al. (1983) .) At this point (beginning of segment 143), we see the subject stumbling and apparently searching around. We call this commonly observed behavior an *impasse*. At the end of segment 143 he proposes `then` as a connective between the loop test and the average computation after the loop. Note that he is unsure whether this is a correct approach (segment 144). We describe this proposed `then` as a patch to the impasse. He tests this usage by sounding the phrase to himself (segment 146), reasoning from Pascal's `if then` statement. Finally, in segment 148, he uses phrasing from Natural Language to justify his usage of the buggy `repeat . . . until then . . . .` At this point he has written the `then` and committed the bug.

Our model draws on the Repair Theory work of Brown and Van Lehn (1980) . They studied impasses in children's subtraction algorithms and developed a detailed theory and process model for how bugs arise and are patched (in their work they refer to "repairs" instead of "patches"). In the more complex domain of programming we have developed a model with two important components:

1. We characterize the knowledge that allows a novice to form a bridge between programming language syntax and semantics, and higher-level design concerns. This is information about how the language constructs are used to accomplish standard programming tasks. We represent this information as schema-like structures called *programming plans*. We discuss the programming plans for both the introductory programming language Pascal and Natural Language step-by-step procedures.

Write a program which repeatedly reads in integers until it reads the integer 99999. After seeing 99999, it should print out the CORRECT AVERAGE without counting the final 99999. Remember, the average of a series of numbers is the sum of those numbers divided by how many numbers there are in the series.

Figure 1: The Averaging Problem.

Subject 13: [reading from problem] After seeing 99999, it should print out the correct average, so until, ahhh, until I equals 99999 [he writes `until I = 99999`].

Interviewer: Ok

Subject 13: Ahhh, and then, [he pauses, reading what he's written] "repeat, until", and then I would have, then [writes `then`].

Subject 13: Can you have a `then` in the `repeat until`? No, I don't recall that.

Interviewer: OK

Subject 13: [mumbles] if then, while then, repeat until then, [louder now] well, it makes sense. I'm not sure if thats right, I don't recall using it before.

Interviewer: How does it make sense?

Subject 13: It makes sense because you are repeating until that [points to `until` line] and *then* [his emphasis] what do you do? Well, *then* take the average. . . . [after the `then` he writes `Average := Sum/N`]

Figure 2: Excerpt of transcript for Subject 13. Up to this point the subject has written a loop body for the Ending Value Averaging problem. Here he is working on specifying the test for the loop and the code following the loop. Square brackets enclose comments we have added.

2. We characterize impasses arising from missing or mis-applied programming plans needed in the course of developing a program. We propose that many bugs arise out of novice strategies for patching an impasse and continuing a problem solution. We call these strategies ***bug generators***. We focus on the bug generators that patch an impasse by using the knowledge of how the problem would be solved with Natural Language step-by-step procedures. This knowledge is used to to supply missing programming language knowledge.

The paper is organized as follows. In section 2 we discuss programming plans as a representation to describe the both the knowledge used to write step-by-step Natural Language procedures and the knowledge used to write Pascal programs. We also describe the relationship between Natural Language and procedure knowledge and Pascal programming knowledge. Section 3 discusses process of novices resolving impasses with patches characterized by our bug generators. In section 4 we present a preliminary evaluation of the model.

2. Representing Novice Programming Knowledge

The content and structure of novice knowledge has been studied in several different domains including physics (Chi et al., 1981; DiSessa, 1982), geometry (Anderson et al., 1981), algebra (Lewis, 1981; Matz, 1982), and programming (Ehrlich & Soloway, 1983; Rich, 1981; Soloway et al., 1982). A key generalization is that novices use schema-like structures we call ***plans***. We propose two kinds of plan knowledge and links between those two kinds of plans:

- Novice knowledge of step-by-step Natural Language procedures - We refer to the set of step-by-step Natural Language procedure plans that a novice brings to a programming course as *SSK*: Step-by-Step Natural Language programming Knowledge.
- Novice knowledge of the programming language under study - We refer to these plans as *PK*: Pascal programming Knowledge.
- Functional and Surface Links between the SSK and PK - These links allow a novice to traverse between the two different sets of knowledge structures.

In what follows, we describe each of these components in turn.

2.1. SSK: Pre-Programming Knowledge

Novices bring knowledge of standard tasks in step-by-step Natural Language procedures to their introductory programming course. Examples of such tasks include: looping, making choices, and specifying sequences of actions. We have studied this pre-programming knowledge with problems like the Factory Gate Problem, shown in Figure 3. The problem asks the subject to

write a step-by-step Natural Language procedure for a junior clerk to collect payroll information from workers coming out of a factory gate. The clerk needs to report on the average salary for all the workers that leave the gate before the first supervisor leaves the gate. This problem was designed to parallel the Averaging Problem.

Figure 4 shows a sample solution to the Factory Gate problem. This solution is from one of our subjects before he had taken a programming course. It illustrates several features of step-by-step Natural Language procedures. Step 5, for example, specifies counting with the phrase "add number of" while step 6 specifies a total with the phrase "add all". The overall structure of the loop (steps 1 to 4) is specified by illustrating how to do the task for the first two workers followed by the phrase "and so on" (step 3). Finally, step 4 specifies the stopping condition for the loop. This condition is phrased as a continuously active test, always watching the action of the loop for the exit condition to become true. (This is sometimes referred to as a "demon" control structure.) In general, the novice seems to be using plans for standard tasks like adding up numbers, stoping the loop, etc.

When coming into a programming course, novices will have a fairly complete SSK. This is to be expected because SSK is regularly used by most people to specify simple step-by-step procedures. Bonar (1985) discusses SSK plans in detail. See Miller (1981) for an earlier discussion of the some standard features in step-by-step Natural Language procedures.

2.2. PK: Fragments of Programming Knowledge

PK represents the knowledge that allows novices to write some parts of a program correctly. Besides the obvious information about syntax and semantics, PK plans also capture the goals and tactics critical to implementing some task as a program. Our plans contain several different pieces of information. Consider, for example, the Counter Variable Plan shown in Figure 5 and its components:

- Plans are linked together in a general/specific hierarchy. The Counter Plan, for example, is a specialization of the Running Total Variable Plan.

- Plans contain information about the key roles and aspects of the plan. In the Counter Variable Plan we represent this information as slots describing the plan, its initialization, its update, how it is used, and the type of the variable.

- Plans contain knowledge needed to actually implement a plan in a specific programming language. With the Counter Variable plan there is specific information about how a Counter is implemented in Pascal.

Please write a set of explicit instructions to help a junior clerk collect payroll information for a factory. At the end of the next payday, the clerk will be sitting in front of the factory gates and has permission to look at employee pay checks. The clerk is to produce the average salary for the workers who come out of the door. This average should include only those workers who come out before the first supervisor comes out, and should not include the supervisor's salary.

Figure 3: The Factory Gate Problem.

1. Identify worker, check name on list, check wages
2. Write it down
3. Wait for next worker, identify next, check name, and so on
4. When super comes out, stop
5. Add number of workers you've written down
6. Add all the wages
7. Divide the wages by the number of workers

Figure 4: Typical answer for the Factory Gate Problem. This procedure was written by Subject 11 before studying Pascal.

Counter Variable Plan

SPECIALIZATION OF ⇒ Running Total Variable Plan

- □ *Description:* Counts the occurences of some specific action.
- □ *Initialization:* Set to zero.
- □ *Update:* Adds one to the current value.
- □ *How used?:* Always found within a piece of code that is executed repeatedly.
- □ *Type:* Integer values.

Pascal Implementation of Counter Variable Plan

- □ *Initialization:* `cv := 0`
- □ *Update:* `cv := cv + 1`
- □ *How used?:* Usually found within a `while` or `repeat` loop. Counts are are created by successive increments as the events to be counted occur.
- □ *Type:* `integer`

Figure 5: Counter Variable Plan. The double right arrow (e.g. ⇒) indicates a link to another (more general) plan. The boxes (e.g. □) indicate slots of the plan.

While a novices' SSK is relatively complete, their PK is fragmentary. This is to be expected, the novice is just learning about PK. Not only do novices have fewer PK plans than would an expert, the connections between the PK plans are less rich.

2.3. Functional and Surface Links Between SSK and PK

While we have discussed SSK and PK separately, they have many similarities. In particular, we have characterized two kinds of similarities:

- Functional similarities exist because both SSK and PK are concerned with repeated actions, choice between conditions, counting, etc.

- Surface similarities exist because the programming language Pascal (like most others) shares many words with Natural Language. There are many common lexical entities in the two plan sets, irrespective of functional similarity of the plans connected.

We capture these similarities as functional and surface links between the SSK and PK plan sets. While we distinguish between these two kinds of links, novices often do not. For example, the bug in the protocol of Figure 2 shows a confusion between surface and functional links. We see that he is confused about the relationship between the Natural Language "then" (indicating a following step) and the `then` of Pascal's `if then else` construct. We say that he is using the surface (lexical) link between "then" and `then`. Based on this surface link, he is assuming a functional link that allows him to use uses `then` with a Natural Language meaning.

Another example from the protocols illustrates confusion between surface and functional links associated with the word "while". In Natural Language "while" is typically used as a continuously active test, as in: "***while*** the highway stays along the coast, keep following it north". This kind of control structure is unusual in a programming language. More typical is a construct where the loop condition gets tested once per loop iteration, e.g. the `while` loop in Pascal. The surface link between the two kinds of "while" allows a novice to infer similar semantics. One of our subjects even inferred a semantics for a continuously active test in the Pascal `while` loop: "every time [the variable tested in the `while` condition] is assigned a new value, the machine needs to check that value . . . ".

3. The Process of Generating a Bug

Since novice programming knowledge is fragmentary, by definition there must be gaps in that knowledge. Thus, in writing a program a novice will encounter such a gap and be at what we have called an impasse. In order to bridge these gaps, we say that the novice uses ***patches***. By their very nature, these patches are likely to be incorrect. For this reason we call the processes that create these patches ***bug generators***. We argue that the bug generators draw on inappropriate or incorrect knowledge such as from SSK.

Consider a detailed example of some bug generators. Subject 13 wrote `Sum := 0 + I` as part of a running total update inside a loop body for the Averaging Problem (see his code in Figure 6). He has identified `I` as the variable to receive new values, entered by the user. He has also identified `Sum` as the variable to hold the running total. Although there are a number of problems with this code, we focus on a single bug. (In the Appendix we show an detailed excerpt from Subject 13's protocol. That excerpt shows Subject 13 actually writing this code and attempts to analyze all the bugs that appear there.) In the protocol, Subject 13 points to this line and says "it reads the Sum". The only `Read` statement Subject 13 has used has no arguments and is above the loop along with a `Readln`. What, then does the subject mean by "reading the sum"? Our bug analysis describes several relevant bug generators and is shown in Figure 7.

The impasse in this case is that Subject 13 did not know how to implement a read operation for an Input New Value Variable in Pascal. A correct Pascal implementation uses an explicit `Read(I)` in each iteration of the loop. Three plausible explanations, each based on a different bug generator, and one based on a slip interpretation are shown for the bug. (The reason for multiple explanations is discussed below.)

1. The first Bug Generator plausibly explaining this bug is the ***Programming Language used as if it were Natural Language*** (*PL Used as NL*) Bug Generator. This bug generator operates on a specific plan or plans, using surface links between SSK and PK versions of the plan. Even though they are surface links, the novice treats them as functional links, assuming that the Natural Language semantics can be used for programming language constructs. The programming language construct is given similar semantics to the parallel Natural Language construct. In the example with

```
N := 0;                          [1]
Sum := 0;                        [2]
repeat                           [3]
    Sum := 0 + I                 [4]
    N := 1                       [5]
    Sum := I + I                 [6]
    N := 2                       [7]
until                            [8]
```

Figure 6: Subject 13's first attempt at the loop for the Averaging Problem. Line numbers are referred to in the protocol shown in the Appendix. `N` is the Counter Variable, `Sum` is the Running Total Variable, and `I` is the Input New Value Variable.

BUG: "It reads the Sum": `Sum := 0 + I`

The subject should be saying something like "it reads in a value which is added into the sum".

☞ ***PL Used as NL* on Input New Value Variable**

The read operation is done implicitly whenever a value is needed.

☞ ***PL Interpreted as NL* on Pascal - `Read/Readln`**

Here he is treating `Read; Readln` pair as if they were declaring that reading is to be done, and the results of the read will be used with Sum.

☞ ***Multi-Role Variable* on Running Total Variable, Input New Value Value**

The subject is using the variable Sum as if it will automatically get a new value added in whenever that new value is read.

☞ ***Slip***

Just slipped and forgot to say ". . . a value which is added . . . ".

Figure 7: Bug analysis showing the operation of bug generators. The right point hands (e.g. ☞) indicate each different plausible bug generator explanation for the bug.

Subject 13, the *PL Used as NL* Bug Generator operates on the *Input New Value Variable* plan. Getting a new value for the *Input New Value Variable* has been implemented implicitly, which we claim is typical of the Natural Language implementation.

This claim about Natural Language implementation derives from our studies of SSK using problems like the Factory Gate Problem of Figure 3. In those studies we found that getting data was often implemented implicitly. For example, in line 1 of Subject 11's answer to the Factory Gate Problem (Figure 4), he says "1. Identify worker, check name on list, check wages." Here, we claim, getting the input value (the worker) has been done implicitly. Note that this operation is not always implicit. In line 3 of the same Factory Gate procedure, for example, the subject says "3. Wait for next worker, identify next, check name, and so on." In this case we say that getting the input value has been done explictly with the phrase "Wait for next worker . . . "

2. The second explanation for the Bug in Figure 7 uses the *Programming Language interpreted as Natural Language* (*PL Interpreted as NL*) bug generator operating on the `Read` statement written above the loop. In this case, the subject is seen to be interpreting the `Read` as if it were a declaration statement. (There is support for this explanation in that the Subject discussed the `Read` statement while discussing other declarations.) The *patch* was to interpret the `Read` as if it were Natural Language. We claim that in Natural Language one can declare that reading will be done, and have that reading be implicit in the rest of the step-by-step procedure.

This claim about Natural Language again derives from our studies of SSK. This notion of an input declaration comes from a Natural Language implementation

strategy where the subject uses an early step in the procedure to describe how new data values are retrieved. In later steps, new input values are referred to implicitly. For example, on of our subjects working on the Factory Gate Problem used the lines: "Each worker will be coming out of the gate in turn. Write down the value of each paycheck . . . "

Note that the distinction between the *PL Used as NL* Bug Generator and the *PL Interpreted as NL* Bug Generator is fine. In the *PL Interpreted as NL* bug generator, the novice uses a programming language construct to implement a Natural Language plan; a `Read` used as a declaration that reading will occur into a certain variable. In the *PL Used as NL* bug generator, on the other hand, the novice uses programming language constructs as if they had their Natural Language meanings or omits programming language constructs that would not be needed in Natural Language: the implicit `Read(I)` every time a data value is needed.

3. The third explanation for the bug in Figure 7 uses the *One variable assumed to have multiple roles* (*Multi-Role Variable*) Bug Generator. In this interpretation the subject patches an impasse about getting new values in the loop by collapsing the purpose for two different variables. In particular, he has collapsed the running total to be accumulated with the new value to be read from a user. In this interpretation, he thinks that the running total happens automatically when a new value is read from the user. The subject understands the statement `Sum := 0 + I` to mean that the variable `Sum` gets the value of `I` added in every time a new value of `I` gets `Read`. This bug generator derives from the flexibility of a Natural Language noun phrase. A noun phrase can have several different aspects, normally disambiguated by context. In the *Multi-Role Variable* interpretation of our example, the noun phrase "the sum" has

the two aspects ***accumulated so far*** and ***made up of values read***. Rosnick (1982) has found a similar collapsing of variable roles in algebra students solving word problems.

4. The fourth explanation is a ***Slip*** Bug Generator explanation. It says that the subject simply spoke sloppily, and clearly understands that a `Read(I)` statement must appear elsewhere.

From the example in Figure 7 we see that it is possible to produce plausible explanations for a bug based on patches from two or more bug generators. If several different patches produce the same result, there is no systematic way to choose between the different possible bug generators. In fact, it is reasonable that novices can construct a patch using several bug generators that suggest similar approaches. Given the current methodology, however, we have no way to relate the novice programmer's behavior to possible interactions between plausible bug generator explanations.

Notice that each bug generator in the example we described in terms of the plans active at the time of the impasse. We view bug generators as procedures that use one or more active plans to create a patch. We have categorized the bug generators based on how they use the active plans. We describe each of these three catagories, with examples of each:

1. ***SSK Confounds PK:*** With these bug generators the novice uses an SSK version of an active plan to confound the PK version of that plan.

   PL Used as NL (Programming Language Used As Natural Language)

   This bug generator uses a programming language construct because it is has the same words as a phrase used in the natural language implementation of an active plan. For example, in Figure 7 Subject 13 may have have used the programming construct `read` as implying an implicit read operation to be executed whenever needed. In Natural Language, data is often retrieved in this way.

PL Interpreted as NL (Programming Language Interpreted as Natural Language)

This bug generator interprets a programming language construct as if it were a phrase used in a Natural Language implementation of the plan on which it is operating. In Figure 2, for example, Subject 13 seems to have interpreted `then` from Pascal as the phrase "then" in Natural Language.

Multi-Role Variable (Multiple Roles For a Variable)

This bug generator uses a single variable for multiple roles from current active plans. This bug is designed to capture that aspect of Natural Language where a single noun phrase can have several different referents depending on context. For example, in Natural Language we can talk about the "sum of the wages" and "the count of the wages". Similarly, one of our subjects seems to have used the same variable to stand for both a Running Total Variable (summing input values) and the Counter Variable (counting those input values). (In the excerpt, `I` is the variable the holds the input values and `N` is the multi-role Counter Variable and Running Total Variable):

"I want to get a statement that is going to be clear that we're going to add the numbers, each number entered, we'll have the tally of the, number of integers entered . . . ahhh, N equals, ahmmm integer. [Writes: `N := I`]".

NL Construct (New Programming Language Construct From Natural Language)

In this bug generator a new programming construct is invented based on a Natural Language implementation of the parameter plan. For example, on of our subjects wrote the following line:

```
New := next New;
```

to indicate that the next value is needed for the New Value Variable.

Generic Name (Variables Named Generically)

This bug generator uses generic names for parts of the program or variables. These names are based on common programming language implementation strategies. Program elements are named based upon how we talk about them when describing the program. For example, our subjects would talk about the ***Input New Value Variable*** as the "the variable holding the integers read in" and give that variable the name `integer`.

2. ***Intra-PK:*** These bug generators use incorrect or incomplete PK versions of the active plan to patch an impasse. Patching occurs by using knowledge within PK.

Trace (Statements Ordered In Execution Order)

This bug generator orders the program as if it is an execution trace. For example, one of our subjects produced the following code for the body of Averaging Problem: (`Sum` is the Running Total Variable, `I` is the New Value Variable, and `N` is the Counter Variable.)

```
Sum := 0 + I
N := 1
Sum := I + I
N := 2
```

Over Generalize (Programming Language Overgeneralization)

This bug generator allows a subject to overgeneralized from one Pascal implementation plan to another. For example, subjects often will initialize all variables whether this is needed or not.

Tactically Similar (Tactical Similarity)

In this bug generator the subject fails to distinguish between plans that do

similar things but are implemented differently. For example, subjects will use the assignment operator to give a value to Pascal constants or in association with a **read** statement.

3. ***Other Confounds PK:*** With these bug generators the novice uses knowledge from other (i.e. not SSK or PK) domains to confound the PK version of the active plan.

Other Domain (Knowledge From Other Domains)

This bug generator uses an understanding from a domain like mathematics. For example, one of our subjects assumed that the variable I will always increment implicitly, much like the *i* used in mathematical notation for a series.

OS Confound (Operating System Confound)

This bug generator confounds a command or operation from the operating system with a programming language construct. For example, one of our subjects used an editor command within his program.

Our bug descriptions also include a ***Slip*** as a plausible explanation. By slips we mean a random error produced while the novice is distracted, a speech slip, or a typographical error.

4. Evaluating the Model

This section presents a preliminary evaluation of the model we present. In particular, we present evidence that a significant portion of novice bugs can be explained by the *SSK Confounds PK* bug generators. Our data is taken from a detailed analysis of four protocols selected from a body of interviews originally conducted to explore novice programming cognition. In the selected protocols we present the same introductory programming problem to four different novice programmers, all in the fourth to sixth week of an introductory programming course.

4.1. Methodology

In our interviews we present introductory programming students with a Pascal programming problem. Subjects are instructed to think aloud as they develop a solution to this problem. Protocols of these sessions were then analyzed using standard techniques used to study novice understanding of physics (Chi et al., 1981) , algebra (Clement, 1982), children's arithmetic (Resnick, 1982), and other domains. We have also drawn on methodologies developed by Newell and Simon (1972) and discussions of methodological issues arising from the use of verbal reports as data (Ericsson & Simon, 1980; Ginsberg et al., 1981).

Our analysis of the protocols has two steps: the ***plan analysis*** and the ***bug analysis***. In the plan analysis we relate the relatively abstract plan descriptions to the actual utterances and programs produced by the novice. In addition, since our claim is that novices use both SSK and PK while programming, we need to know whether the novice has used an SSK or PK implementation of each plan. In order to make the distinctions between SSK and PK versions of each plan, we have developed a series of criteria for each. The criteria specify those subject utterances we use to justify a claim that a certain plan is in use.

The criteria for SSK plans are based on our analysis of a study where 34 subjects were given problems like the Factory Gate Problem (Figure 3). Subjects needed to use some implementation for each plan needed to solve a problem. Each implementation found (including the case of not explicitly using the plan) is represented on the list of criteria. These lists of criteria represent the strategies we know to be used in implementing each plan. The criteria for PK plans are based on our own introspection about the implementation strategies for PK plans.

Consider the example of plan analysis and use of criteria in Figure 8. In that plan analysis we claim that Subject 6 has used both an SSK and PK version of the *Results Variable* plan. In segment 173 she is discussing her implementation of the *Result Variable*. In the first part of the segment she describes the implementation without any reference to how the count and sum are accumulated. This satisfys a criterion for an SSK version of the plan. Later in the segment, she goes into detail about the accumulation of the count and sum, satisfing a criterion for a PK version of the plan. We call each such piece of evidence (i.e. satifaction of a criterion) an ***evidence item***.

A second example in Figure 9 illustrates how subtle the plan analysis can get. In the plan

Subject 6: ". . . what I'll want to be doing at the end is dividing the sum by the total of numbers to find an average, so I'll always have to keep track of both. So, after each number is read in, I will have, I will be keeping track at that same time as the calculation is made of the count and of the sum, so at any one point where the sentinel's read in, I will have both figures available to read in an average."

PLAN: Result Variable
Evidence: Natural Language
". . . dividing the sum by the total of numbers to find an average, . . . "
Evidence: Pascal
"after each number is read in, . . . I will be keeping track . . . of the count and the sum . . . "

Figure 8: Protocol segment with claimed use of both a SSK and PK version of the *Results Variable* Plan.

analysis we claim that Subject 11 has used both an SSK and PK version of the *Counter Variable* plan in a single sentence (segment 112). We offer evidence for each version:

1. the SSK version of the *Counter Variable* plan: the phrase "counting the number of integers that come through".

2. the PK version of the *Counter Variable* plan: the phrase "implementing by ones".

These evidence items suggest that the novice is using ***both*** the SSK and PK versions of the plan within the same sentence. The difference is based on how the counting operation is viewed. "Counting the number of integers that come through" indicates that the subject intends an operation that counts the elements of a set that has been completely acquired before the count starts. On the other hand, "incrementing by ones", both uses programming jargon ("incrementing") and indicates an operation that counts each element, one at a time.

Counts of evidence items are used in our quantitative summaries (presented below) to represent overall plan activity. Note that a plan instance with several evidence items actually counts once for each evidence item. In our example of Figure 9 we would count two evidence items, one for an SSK version of the plan and one for a PK version of the plan.

In the ***bug analysis*** we show how errors made by a novice can be plausibly explained as bug generators operating on currently active plans. We analyze each piece of buggy behavior in a separate bug annotation. In Figure 7 we showed an example of such an annotation. After briefly describing the bug, the annotation describes the bug generators and plans that plausibly explain the bugs.

The bug generator set used in our bug analysis was developed after examination of the first protocols presented here. That bug generator set was then used to analyze the other three protocols. In this way, we provide a minimal test of the explanatory power of chosen set.

4.2. Overview of Protocols Analyzed

Before presenting a detailed analysis of the subjects performance, we present an overview of each subject's work on the Averaging Problem. These programs and descriptions provide a context for the data that follows. (Superscript numbers, e.g. ^{1}, ^{2}, etc., in the figures showing the subject's programs are used to refer to lines of code discussed in the text.)

Interview 1:

Subject 11: ". . . I want it simply incrementing by ones, it's counting the number of integers that come through, . . . "

PLAN: Counter
Evidence: Natural Language
"counting the number of integers . . . "
Evidence: Pascal
"incrementing by ones"

Figure 9: Protocol segment with claimed use of both a SSK and PK version of the *Counter Variable* Plan.

Subject 13's final program is shown in Figure 10. There are a number of things wrong with this program, but most critical is his peculiar loop body. Notice that within the loop body, `Sum` (the Running Total Variable) is first set to `0` (see 1), and then to `I + Next I` (`I` is the Input New Value Variable) (see 2), while `N` (the Counter Variable) is set to `1` (see 3) and then `2` (see 4). The subject seems to be implementing the loop with the following Natural Language strategy: ***Show an example of the first few steps, and assume that the other iterations will happen correctly*** (he actually shows the first two steps). In the Appendix we show an analyzed excerpt from Protocol 1.

Interview 2:

Subject 6's The final program is shown in Figure 11. It is almost correct. His problem is that there is no `READ` inside the loop. In the protocol he convinces himself that he needs a `READ` above the loop (see 1) to make the `WHILE` test make sense (see 2), but never thinks to also put a second `READ` inside the loop. At one point, he uneasily states that each test of `NEWNUM` in the `WHILE` statement (see 2) will know to read a new value for that iteration.

Interview 3:

Subject 11's final program is shown in Figure 12. Her program is almost completely correct. Like Subject 6, her bugs are related to reading new values inside the loop. She recognized that some sort of `Read` was required inside the loop, and even realized that it had to come after the `Count` increment (see 1) and `Total` update (see 2). The `Read (I)` below the loop (see 3) was originally, and correctly, put at the bottom of the loop body. In the protocol she argues that leaving it there would cause the program to read a new value before the previous value read is processed. She uneasily moved `Read (I)` out to it's current position outside the loop (see 3) and settled on `Readln` (see 4) at the bottom of the loop to express, as she put it, "the right amount of reading".

```
Program Average (Input/, Output);
Var
   N, Sum : Integers
   Average : Real;
Const : Sentinel
Begin (* Average of the integers entered *)
   Writeln ('Enter series of integers to be averaged');
   Writeln (Integers will be Averaged when you
                                        enter the Integer 99999);
   Read;
   Readln;
   Sentinel := 99999;
   N := 0;
   Sum := 0;
Repeat
   Read;
   Readln;
   Sum := 0                                                    [1]
   N := 1                                                      [3]
   Sum := I + Next I                                           [2]
   N := 2                                                      [4]
   until I = 99999
   then Average = Sum/N
Writeln ('Average':= 0);
END.
```

Figure 10: Final program for Subject 13 working on the Averaging problem. Numbers mark lines dis- issed in the text.

BEST COPY AVAILABLE

```
PROGRAM AVERAGE (INPUT/, OUTPUT);
CONST
   SENT = 99999;
VAR
   NEWNUM, COUNT, SUM, AVE : INTEGER;
BEGIN
   COUNT := 0;
   SUM := 0;
   READLN;
   READ (NEWNUM);                                          [1]
   WHILE NEWNUM <> SENT                                    [2]
     DO BEGIN
        SUM := NEWNUM + SUM;
        COUNT := COUNT + 1;
        END;
   IF COUNT <> 0
      THEN AVE := SUM DIV COUNT
   WRITELN ('AVERAGE = '), AVE:8:2;
END.
```

Figure 11: Final program for Subject 6 working on the Averaging problem. Numbers mark lines discussed in the text.

BEST COPY AVAILABLE

```
Const
   Sentinel = 99999;
Var
   I, Count, total, AVG : Integers;
Begin
   Count := 0;
   Total := 0;
   Writeln ('Enter integer');
   Readln;
   Read (integer);
   While I <> 99999 Do
      Begin
         Count := Count + 1;                    [1]
         Total := Total + I                     [2]
         Readln                                 [4]
      End
   Read (I)                                     [3]
   Avg := Total Div Count
   Writeln ('Avg is ', Avg:0)
```

Figure 12: Final program for Subject 11 working on the Averaging Problem. Numbers mark lines discussed in the text.

BEST COPY AVAILABLE

Interview 4:

Subject 12's final program is shown in Figure 13. This program is almost correct. It's bug involves the first value READ (before the start of the loop) (see 1). She reads in a starting value, but then sets NUM (the Input New Value Variable) to 0 (see 2), losing that starting value. From the protocol, it seems that she does this because she wanted to initialize all variables used inside the loop to 0.

Figure 14 contains overview statistics for the four protocols analyzed. Interviews lasted between 23 and 45 minutes. It is not clear why subjects 2 and 4 took nearly twice as long as subjects 1 and 3. Time spent on the problem does not correlate with plan usage or with number of bugs.

5. Results of the Analysis

We present our analysis based on three expectations derived from our model of novice programming bugs.

5.1. Expectation 1: Regular and Extensive Use of Plans

We have claimed that novices use both SSK and PK represented as plans. If this is the case then plan usage should be pervasive:

> **Expectation 1:** Novices show a regular and pervasive use of plans. Novices should make use of ***both*** SSK and PK plans.

Figure 14 contains the statistics relevant to expectation 1. There were 104 plan evidence items in the protocol with the fewest plan evidence items (protocol 3). This protocol averaged 4.5 plan evidence items per minute. The protocol with the most evidence items had 177, averaging 3.9 plan evidence items per minute. While there is no baseline, it seem fair to say that our subjects did use plans pervasively.

In Figure 15 we show the plan evidence item counts broken down by type of plan (with SSK and PK lumped together). From this fine-grained breakdown we see that all subjects used every type of plan. More intriguing, however, is that this breakdown permit an interesting speculation. Consider the following data about several different plans:

BEST COPY AVAILABLE

```
PROGRAM SUMUP (INPUT/,OUTPUT);
CONST
   SENTINEL = 99999
VAR
   NUM1, SUM, AVERAGE, COUNT : INTEGER;
BEGIN
   WRITELN ('READ IN AN INTEGER AND CONTINUE UNTIL');
           ('FINISHED THEN ENTER 99999 . . . ');
   READLN;
   READ (NUM);                                                    [1]
   COUNT := 0;
   SUM := 0;
   NUM := 0;                                                      [2]
   WHILE NUM <> SENTINEL DO
       BEGIN
       SUM := NUM + SUM;
       COUNT := COUNT + 1;
       READLN;
       READ (NUM)
       END;
AVERAGE := SUM DIV COUNT;
WRITELN ('AVERAGE IS, ' AVERAGE : 0);
END.
```

Figure 13: Final program for Subject 12 working on the Averaging Problem. Numbers mark lines discussed in the text.

	Protocol Number			
	1	2	3	4
Subject Number	13	6	11	12
Duration (minutes)	32	45	23	45
Plan Evidence Items	159	177	104	107
SSK Plans	43	50	16	16
PK Plans	116	127	88	91
Plan Evidence Items/Minute	5.0	3.9	4.5	2.4
Bugs[1]	32	19	11	10

Figure 14: Summary statistics for the four protocols analyzed. [1]"Bugs" refers to the number of bugs found in the protocol.

New Value Controlled Loop Plan

In protocol 1, where the subject's loop was very far from correct, the count is quite low, accounting for only 6% of the plan evidence items. Other subjects, whose loops were fairly close to correct, had higher counts ranging between 13% and 21% of all plan evidence items.

Sentinel Variable Plan

All subjects had little trouble with this plan and all had relatively low plan evidence item counts, ranging between 4% and 10% of all plan evidence items.

Input New Value Variable Plan and *Input Plan*

These plans gave all subjects a great deal of trouble. Plan evidence items counts (for the two together) range between 18% and 33% of all plan evidence items.

From these and similar observations, it appears that plan activity is highest on those plans that a novice *almost* knows. That is, those plans where the novice has a fairly good idea of what is needed, but does not have all aspects correct.

5.2. Expectation 2: Bug Generators Plausibly Explain the Bugs

Expectation 2: The Bug Generator set presented here can plausibly explain most errors found in the protocols.

The data relevant to this expectation appear in the "Bugs Explained" line of Figure 16. As discussed earlier, the bug generator set was developed based on the analysis of protocol 1. The data indicate that one or more bug generators from that set can plausibly explain each of the bugs found in protocols 2, 3, and 4.

It is important that the bug generators actually do discriminate between the bugs. In our data, this would mean that there are relatively few plausible explanations per bug. Figure 16 shows the total number of plausible explanations in each protocol and the mean number of plausible explanations per bug. These data indicates that while multiple bug generators may be responsible for some bugs, on the average less than two bug generators can plausibly explain each bug.

The plausible bug generator explanations summarized in Figure 16 do not include cases

	Protocol Number							
	1		2		3		4	
Subject	13		6		11		12	
Plan Evidence Items		(%)		(%)		(%)		(%)
Sentinel	9	(5%)	8	(4%)	4	(4%)	11	(10%)
Loop	20	(13%)	13	(7%)	9	(9%)	1	(1%)
Counter	29	(18%)	21	(12%)	18	(17%)	12	(11%)
Arithmetic Sum	21	(13%)	38	(21%)	13	(13%)	15	(14%)
Result	23	(14%)	18	(10%)	9	(9%)	12	(11%)
New Value	11	(7%)	13	(7%)	16	(15%)	16	(15%)
Input	18	(11%)	21	(12%)	15	(14%)	8	(7%)
New Value Loop	9	(6%)	35	(19%)	14	(13%)	23	(21%)
Result Output	10	(6%)	6	(3%)	0	(0%)	1	(1%)
Instructional Output	7	(4%)	0	(0%)	3	(3%)	0	(0%)
Prompt Output	2	(1%)	1	(1%)	3	(3%)	8	(7%)
Illegal Filter	0	(0%)	3	(2%)	0	(0%)	0	(0%)
Total	159	(100%)	177	(100%)	104	(100%)	107	(100%)
Bugs	32		19		11		10	

Figure 15: Breakdown of plan usage statistics for the analyzed protocols.

	Protocol Number			
	1	2	3	4
Subject Number	13	6	11	12
Bugs Found	32	19	11	10
Bugs Explained[1]	-	19	11	10
Total Explanations[2]	55	27	15	15
Explanations/Bug[3]	1.7	1.4	1.4	1.5

Figure 16: Bug generator statistics for the analyzed protocols. [1]"Bugs Explained" refers to the number of bugs that can be explained by one or more bug generators developed from the analysis of protocol 1. [2]"Total Explanations" refers to the number of plausible, non-slip, bugs generator explanations. [3]"Explanations/Bug" refers to the average number of plausible, non-slip explanations per bug.

where the bug was plausibly explained as a slip. These data are summarized in Figure 17. First, notice that all but two bugs had a plausible non-slip explanation. That is, almost every bug detected in the analysis can be plausibly explained as a bug generator operating on plans. Second, notice that slips can plausibly explain between 34% and 58% of the bugs (depending on the protocol). Most likely a much smaller percentage are actually slips. Though there are many bugs that can plausibly be interpreted as slips, many of these slip interpretations become less likely when the bug is examined in the context of the whole protocol. That is, subjects often return to a bug and explicitly discuss it. It is unlikely that a slip would stand up to such repeat scrutiny.

5.3. Expectation 3: SSK/PK Bug Generators Are Critical

In our model we propose that the intrusion of SSK plays an important role in novice programming bugs:

> **Expectation 3:** When novices encounter an impasse in a developing programming solution, they usually use *SSK Confounds PK* bug generators to patch and continue.

The analysis presented here contains two kinds of evidence for this expectation.

- *SSK Confounds PK* Bug Generators embody the process whereby novices use SSK knowledge to reason about their programs. As can be seen in the "SSK/PK Total" line of Figure 18, *SSK Confounds PK* Bug Generators plausibly explained 60%, 67%, 67%, and 47% of the bugs found in protocols 1 to 4 respectively. That is, for protocols 1 to 3, *SSK Confounds PK* Bug Generators can explain 60% or more of the bugs. Protocol 4 had the lowest (47%) coverage by *SSK Confounds PK* Bug Generators but also had the least buggy protocol.

- The subjects had between 10 and 32 bugs. Notice that in terms of bugs, there seem to be two groups in the protocols: Protocols 1 and 2 have more bugs then protocols 3 and 4. We refer to these as the "buggy group" and the "less-buggy group", respectively. Referring to Figure 19, we see that the buggy group had higher ratios of SSK to PK plan evidence items then the less-buggy group. The ratios were .37 and .39 for the buggy group protocols and .18 for both protocols of the less-buggy group. This is consistent with the notion that a more error prone and less advanced novice is doing more reasoning from SSK plans.

6. Concluding Remarks

In the preceding section we presented an analysis of four protocols of novice programmers. From that analysis we provide data to support our model of novice programmer bugs. Specifically, we present evidence that:

- Plans, both SSK and PK, are pervasive in the work of novice programmers.

	Protocol Number							
	1		2		3		4	
Subject Number	13		8		11		12	
Bugs Found	32		19		11		10	
Bugs Plausibly Explained by Non-Slip Bug Generators	32	(%) (100%)	18	(%) (95%)	10	(%) (91%)	10	(%) (100%)
Bugs Plausibly Explained by Slip Bug Generators	11	(34%)	11	(58%)	5	(45%)	5	(50%)

Figure 17: Slip vs. Non-Slip bug generator statistics.

	Protocol Number							
	1		2		3		4	
Subject Number	13		8		11		12	
Bugs Found	32		19		11		10	
Plausible Explanations By Non-Slip Bug Generators								
		(%)		(%)		(%)		(%)
SSK Confounds PK:								
PL Used as NL	10	(18%)	6	(22%)	1	(7%)	1	(7%)
NL Interprets PL	11	(20%)	10	(37%)	7	(47%)	3	(20%)
Multi-role	8	(15%)	1	(4%)	1	(7%)	3	(20%)
NL Construct	2	(4%)	0	(0%)	0	(0%)	0	(0%)
Generic Name	2	(4%)	1	(4%)	1	(7%)	0	(0%)
SSK/PK Total	**33**	**(60%)**	**18**	**(67%)**	**10**	**(67%)**	**7**	**(47%)**
Intra-PK:								
Trace	10	(18%)	5	(19%)	1	(7%)	1	(7%)
Overgen	3	(7%)	1	(4%)	2	(13%)	3	(20%)
Similar	4	(7%)	1	(4%)	2	(13%)	3	(20%)
Intra-PK Total	**18**	**(32%)**	**7**	**(26%)**	**5**	**(33%)**	**7**	**(47%)**
Other Confounds PK:								
Other Domain	3	(5%)	2	(7%)	0	(0%)	1	(7%)
OS Confound	1	(2%)	0	(0%)	0	(0%)	0	(0%)
Other/PK Total	**4**	**(7%)**	**2**	**(7%)**	**0**	**(0%)**	**1**	**(7%)**
Non-Slip Total	**55**	**(100%)**	**27**	**(100%)**	**15**	**(100%)**	**15**	**(100%)**

Figure 18: Bug generator statistics by bug generator type. The number of bugs plausibly explained by each Non-Slip bug generator. (%) is the percentage of plausible explanations by the specified bug generator out of all plausible Non-Slip explanations. Each bug can have several explanations. Slip data are presented in Figure 17.

	Protocol Number			
	1	2	3	4
Subject Number	13	6	11	12
Plan Evidence Items	159	177	104	107
SSK Plans	43	50	16	16
PK Plans	116	127	88	91
Ratio: PK to SSK	.37	.39	.18	.18
Bugs	32	19	11	10

Figure 19: PK to SSK plan usage ratios support the importance of SSK in understanding buggy programs.

- Our Bug Generator set, developed after studying only one protocol, can plausibly explain the novice bugs we have seen in the other protocols.

- Most importantly, SSK plays an important role in the bugs of novice programmers. Bug generators that describe the confounds between SSK and PK can plausibly explain a substantial number of bugs in each of the protocols. In addition, the SSK to PK plan ratio was higher for those novices with buggier protocols.

There are several next steps planned for this work. Currently, our bug analysis does not provide data to support our specific set of bug generators nor does it provide the ability to recognize which of the plausible bug generators actually contribute to the bug We would like to develop the theory so that this is more specific: are there ways to detect exactly which bug generator(s) is (are) responsible? This will require a more formal definition of each bug generator along with a more constrained set of interviews. In these interviews subjects will be put in specific situations where bugs are very likely. For example, subjects might be given a problem statement and a skeleton program missing a key section. Such interviews would be accompanied by probes to detect the use of certain bug generators and plans. We plan to particularly focus on the Input New Value Variable plan's use within a loop, an area of difficulty for all our subjects. In general, correctly updating within the loop seems to be difficult for our subjects and amenable to more constrained study.

We are planning to explore the use of SSK plans in better understanding the intensions of novice programmers. We are beginning the development of a curriculum organized around common programming plans. Each programming plan would be introduced by examining the SSK version. The SSK plans allow us to understand and direct the students in their basic approach to the problem solution. With the SSK plan providing a connection to something the student already knows, we then introduce the PK version of the plan and actual code. By making a clear distinction between the SSK and PK versions, we should be able to avoid many of the errors discussed in this article.

We have presented a process model of bugs for novice programmers early in a programming course. In developing our model we have focused on the knowledge used by a novice programmer. In particular, our model is based on the role of Natural Language step-by-step procedural knowledge in understanding these bugs. Although our model is just a start toward understanding the source of novice bugs, it indicates how critical novice pre-programming knowledge is likely to be in a more complete theory.

Acknowledgments

The authors wish to thank the referees for their valuable comments and careful reading.

Bibliography

[Anderson et al. 81]
Anderson, J. R., Greeno, J. G., Kline, P. J., & Neves, D. M.
Acquisition of Problem-Solving Skill .
In Anderson, John R. (editor), *Cognitive Skills and Their Acquisition*, chapter 6, pages 191-230. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1981.

[Anderson et al. 84]
Anderson, J. R., Farrell, R., & Sauers, R.
Learning to Program in LISP.
Cognitive Science 8:87-129, 1984.

[Bonar 85] Bonar, J. G.
Understanding the Bugs of Novice Programmers.
PhD thesis, University of Massachusetts, 1985.

[Brown and VanLehn 80]
Brown, J. S., & VanLehn, K.
Repair Theory: A Generative Theory of Bugs in Procedural Skills.
Cognitive Science 4:379-426, 1980.

[Chi et al. 81] Chi, M. T., Feltovich, P., & Glaser, R.
Categorization and Representation of Physics Problems by Experts and Novices.
Cognitive Science 5:121-152, 1981.

[Clement 82] Clement, J.
Students' Preconceptions in Introductory Mechanics.
American Journal of Physics 50:66-71, January, 1982.

[DiSessa 82] DiSessa, A. A.
Unlearning Aristotelian Physics.
Cognitive Science 6:37-76, 1982.

[Ehrlich and Soloway 83]
Ehrlich, K., & Soloway, E. M.
An Empirical Investigation of the Tacit Knowledge in Programming.
In Thomas, John and Schneider, Michael L. (editors), *Human Factors in Computer Systems*. Ablex, Inc., 1983.

[Ericsson and Simon 80]
Ericsson, K. A., & Simon, H.
Verbal Reports as Data.
Psychological Review 87:215-251, 1980.

[Ginsberg et al. 81]
Ginsberg, H. P., Kossan, N., Schwartz, R., & Swanson, D.
Protocol Methods in Research on Mathematical Thinking.
In Ginsberg, H. (editor), *Development of Mathmematical Thinking*, chapter 1. Academic Press, 1981.

[Johnson et al. 82]
Johnson, W. L., Draper, S., & Soloway, E. M.
Classifying Bugs is a Tricky Business.
In *Proceedings of the Seventh Annual NASA/Goddard Workshop on Software Engineering*. NASA, 1982.

[Kahney and Eisenstadt 82]
Kahney, H., & Eisenstadt, M.
Programmers' Mental Models of Their Programming Tasks: The Interaction of Real-World Knowledge and Programming Knowledge.
In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*. Cognitive Science Society, 1982.

[Lewis 81] Lewis, Clayton.
Skill in Algebra.
In Anderson, John R. (editor), *Cognitive Skills and Their Acquisition*, chapter 3, pages 85-110. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1981.

[Matz 82] Matz, M.
Towards a Process Model for High School Algebra Errors.
In Sleeman, Derek and Brown, John Seely (editors), *Intelligent Tutoring Systems*. Academic Press, London, 1982.

[Miller 81] Miller, L. A.
Natural Language Programming: Styles, Strategies, and Contrasts.
IBM Systems Journal 20:184-215, 1981.

[Newell and Simon 72]
Newell, A., & Simon, H. A.
Human Problem Solving.
Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[Resnick 82] Resnick, L. B.
Syntax and Semantics in Learning to Subtract.
In Carpenter, T., Moser J., and Romberg T. (editors), *Addition and Subtraction: A Cognitive Perspective*. Erlbaum, Hillsdale, NJ, 1982.

[Rich 81] Rich, C.
Inspection Methods in Programming.
Technical Report AI-TR-604, MIT, 1981.
MIT AI Lab.

[Rosnick 82] Rosnick, P.
Student Conceptions of Semantically Laden Letters in Algebra.
Technical Report, University of Massachusetts, Amherst, Cognitive Development Project, 1982.

[Shneiderman 76]
Shneiderman, B.
Exploratory Experiments in Programmer Behavior.
International Journal of Computer and Information Sciences 5:123-143, 1976.

[Soloway and Ehrlich 84]
Soloway, Elliot M., and Ehrlich, Kate.
Empirical Studies of Programming Knowledge.
IEEE Transactions of Software Engineering SE-10:595-609, 1984.

[Soloway et al. 82]
Soloway, E. M., Ehrlich, K., Bonar, J. G., & Greenspan, J.
What Do Novices Know About Programming?
In Shneiderman, Ben, and Badre, Albert (editors), *Directions in Human-Computer Interactions*. Ablex Publishing Company, 1982.

[Soloway et al. 83]
Soloway, E. M., Bonar, J. G., & Ehrlich, K.
Cognitive Strategies and Looping Constructs: An Empirical Study.
Communications of the Association For Computing Machinery :853-860, 1983.

I. A Sample of an Analysed Protocol

This appendix contains a short segment from a fully analyzed protocol. This segment is taken from protocol one in the set of four protocols analyzed in depth and discussed in Section 5.

The subject of this segment is working on the Averaging Problem. At the point we pick up the protocol, he is coding the loop. The transcript below is annotated with the plan analysis and the bug analysis. In the course of the protocol he will write the code shown in Figure 6. The subject's basic problem is that he does not understand how variables are used to process values inside the loop. As we will see in the protocol, the subject clearly understands that the loop body must accumulate a running total in the variable `sum` and a count in the variable `N`. He also understands that the variable `I` will hold new values read from the user. Explanations for the peculiar code in the loop body are presented in the bug analysis annotations with the transcript segment shown below.

The transcript uses the following notational conventions:

- Square brackets (e.g. [ ]) enclose explanatory comments we have added to the text.
- Programming font (e.g. `N := 1`) is used to indicate code that the subject actually wrote during the protocol.

- Plan annotations are set off from the protocol text with horizontal lines and labeled "PLAN:". These annotations mark those places where we claim the subject has used a plan. The annotation describes our evidence for that claim. See Section 4.1 for a detailed discussion of the plan annotations.

- Bug annotations are set off from the protocol text with horizontal lines and labeled "BUG:". These annotations mark those places where the subject has a bug. The annotation provides a general explanation of the bug and describes the bug generators that could plausibly explain that bug. Each plausible bug generator explanation is marked with a right pointing hand (e.g. ☞). See Section 4.1 for a detailed discussion of the bug annotations.

Here is the protocol segment:

Subject 13: Ummm, ummm, well I think its, ahh, its [N := I, eventually changed to N := 1, eventually put on line 5] not right I don't think, but I I'm gonna leave it that way for the moment.

Interviewer: OK, Fine

Subject 13: And then integer [the way the subject refers to the variable I], or rather, sum equals integer, ahm, equals zero plus integer, [Writes: Sum := 0 + I, eventually put on line 4] and the number equals the integer, ahhh

---

PLAN: Arithmetic Sum Variable
Evidence: P3 - Running total assignment to Sum in loop.

BUG: Arithmetic Sum set to current New Value Variable
There will be no running sum operation, since the Arithmetic Sum, Sum, always gets the value of the current New Value Variable, I.

☞ ***Trace* on Arithmetic Sum Variable**
The subject is reasoning about the execution behavior of the Arithmetic Sum Variable. He recognizes that on the loops first iteration, Sum will be given the value 0 + I and then writes that.

Interviewer: Why don't you tell me what you are thinking

Subject 13: Well I'm thinking that [points to `N := I`, eventually changed to `N := 1` and eventually put on line 5] should go after [points to `Sum := 0 + I`, eventually put on line 4] because the Sum is going to be zero plus the integer and then the Number is going be, ahh, Number equals 1.

Interviewer: Ah huh

Subject 13: [Writes: [Crosses off `N := I`] And then Number equals one [Writes: `N := 1` on line 5]

PLAN: Counter Variable
Evidence: N7 - Counting to `N` done after operation.

BUG: Counter needs to go after the Arithmetic Sum
The increment of the Counter Variable, `N`, needs to be done after the update of the Arithmetic Sum Variable, `Sum`.

☞ ***PL Interpreted as NL* on Counter Variable**
In our study of Natural Language step-by-step procedures we found that the counting operation is usually specified after the operation to be counted. Also, in one of our interviews, the subject suggested that if counting happened before the operation to be counted, there was a chance that the count would be one too high. Here the subject is applying that convention to Pascal.

Interviewer: OK

Subject 13: And then, and then Sum equals integer plus integer, [Writes: `Sum := I + I` on line 6] and number equals 1. Ahhh

---

PLAN: Arithmetic Sum Variable
Evidence: P3 - Running total assignment to `Sum` in loop.

BUG: Arithmetic Sum set to current New Value Variable

There will be no running sum operation, since the Arithmetic Sum variable `Sum` always gets twice the value of the current New Value Variable, `I`.

☞ ***Trace* on Arithmetic Sum Variable**

The subject is reasoning about the execution behavior of the Arithmetic Sum Variable. He recognizes that on the loops second iteration, `Sum` will be given the value `I + I` and then writes that.

☞ ***Multi-Role Variable* on New Value Variable**

The subject allowing the New Value Variable to take two roles: its "sum so far" and its next value. Note that later in the transcript he will add "next" as a keyword in front of the second `I`.

Interviewer: What are you thinking now?

Subject 13: Number equals 2 [Writes: `N := 2` on line 7] and it would go on, it would repeat, that, if [the loop body] continues to repeat [sweeping motions] this [points to the 2 on line 7] will increase.

PLAN: Counter Variable

Evidence: P3 - Increment counter variable `N` inside the loop

---

PLAN: Indefinite Loop

Evidence: N1 - "Continues"
Evidence: N2 - "this will repeat until . . . "

BUG: Says Counter Variable will increase, but it won't

Says that the Counter Variable, `N`, will increase based on the `N := 1` and `N := 2` inside the loop.

☞ ***PL Interpreted as NL* on Pascal - `repeat`, Indefinite Loop**

The subject is expecting the loop to work like loops in Natural Language. There, it is common to specify a loop by giving one or two cases of the iteration and assuming that person reading will know how to generalize. Notice below () he says that `Sum := 0 + I` (line 4) is the "first format of that" Referring to the action performed for each value of the New Value Variable.

I'm assuming for the moment that this is sufficient input.

Interviewer: OK, "sufficient input"?

Subject 13: Input to [pause] so that the computer will know that, for each [pause] for each integer entered, you add 1, you add the integer to the sum [points to `Sum := 0 + I` on line 4], and that this is the first format of that, zero plus integer, `N` equals 1, sum equals integer plus integer, number = 2, ["next" motion with hand] until [pause] [Writes: `until`]

PLAN: Indefinite Loop
Evidence: N3 - "for each . . . "

PLAN: Counter Variable
Evidence: P3 - Increment Counter `N` inside the loop

PLAN: Arithmetic Sum Variable
Evidence: P3 - Update Arithmetic Sum, `Sum`, inside the loop

This concludes the example protocol analysis.